Logical Verification 2019–2020
Vrije Universiteit Amsterdam
Lecturers: dr. J. C. Blanchette and drs. A. Bentkamp

Repeat Exam
4 February 2020, 18:30–21:15, NU-2B-11
6 questions, 90 points
Answers may be given in English or Dutch

Proof Guidelines

We expect detailed, rigorous, mathematical proofs, but we do not ask you to write Lean proofs. You are welcome to use standard mathematical notation or Lean structured commands (e.g., `assume`, `have`, `show`, `calc`). You can also use tactical proofs (e.g., `intro`, `apply`), but then please indicate some of the intermediate goals, so that we can follow the chain of reasoning.

Major proof steps, including applications of induction and invocation of the induction hypothesis, must be stated explicitly. For each case of a proof by induction, you must list the **inductive hypotheses** assumed (if any) and the goal to be proved. Minor proof steps corresponding to `refl`, `simp`, or `linarith` need not be justified if you think they are obvious (to humans), but you should say which key lemmas they follow from.

You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate, especially for functions and predicates that are specific to an exam question.

In Case of Ambiguities or Errors in an Exam Question

The staff present at the exam has the lecturers' phone numbers, in case of questions or issues concerning a specific exam question. Nevertheless, we strongly recommend that you work things out yourselves, stating explicitly any ambiguity or error and explaining how you interpret or repair the question. The more explicit you are, the easier it will be for the lecturers to grade the question afterwards.

Question 1. Error monad (4+4+5 points)

The *error monad* is a monad stores either a value of type α or an error of type ε. This corresponds to the following type:

```
inductive error (α ε : Type) : Type
| good {} : α → error
| bad  {} : ε → error

export error (good bad)
```

The error monad generalizes the option monad seen in the lecture. The `good` constructor, corresponding to `some`, stores the current result of the computation. But instead of having a single bad state `none`, the error monad has many bad states of the form `bad e`, where `e` is an "exception" of type ε.

1a. Complete the Lean definitions of the `pure` and `bind` operations:

```
def error.pure {α ε : Type} : α → error α ε

def error.bind {α β ε : Type} : error α ε → (α → error β ε) → error β ε
```

1b. Assume `ma >>= f` is syntactic sugar for `error.bind ma f`. Prove the following two monadic laws. Your proofs should proceed step by step, with at most one case distinction, rewrite rule, or definition expansion per step, so that we can clearly see what happens.

```
lemma error.pure_bind {α β ε : Type} (a : α) (f : α → error β ε) :
  (error.pure a >>= f) = f a

lemma error.bind_pure {α ε : Type} (ma : error α ε) :
  (ma >>= error.pure) = ma
```

1c. Define the following two operations on the error monad, using Lean syntax:

```
def error.throw {α ε : Type} : ε → error α ε

def error.catch {α ε : Type} : error α ε → (ε → error α ε) → error α ε
```

The `throw` operation raises an exception `e`, leaving the monad in a bad state storing `e` (i.e., `bad e`).

The `catch` operation can be used to recover from an earlier exception. If the monad currently is in a bad state storing exception `e`, `catch` invokes some exception-handling code (the second argument to `catch`), passing `e` as argument; this code might raise a new exception. If `catch` is applied to a good state, nothing happens—the monad remains in the good state.

Question 2. Logical foundations and mathematics (4+3+6+4+2 points)

2a. What are the types of the following expressions?

```
[true, false, true] : _     option ℕ : _     ℤ : _     Sort 5 : _
```

2b. The function `max` : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ returns the larger one of two given natural numbers (or either if they are equal). Complete the fourth case in the following Lean definition:

```
def max : ℕ → ℕ → ℕ
| 0            0            := 0
| 0            (nat.succ b) := nat.succ b
| (nat.succ a) 0            := nat.succ a
| (nat.succ a) (nat.succ b) := _
```

2c. The type class `add_monoid` is defined as follows in Lean:

```
class add_monoid (α : Type) :=
(add       : α → α → α)
(zero      : α)
(add_assoc : ∀a b c : α, add (add a b) c = add a (add b c))
(zero_add  : ∀a : α, add zero a = a)
(add_zero  : ∀a : α, add a zero = a)
```

Complete the following instantiation of $\mathbb{N}$ with the operator `max` as an `add_monoid` by providing a suitable definition of `zero` and proofs of the `zero_add` and `add_zero` properties:

```
instance monoid_max : add_monoid ℕ :=
{ add       := max,
  zero      := _,
  add_assoc := sorry,
  zero_add  := _,
  add_zero  := _ }
```

2d. The function `min` : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ returns the smaller one of two given natural numbers. Can $\mathbb{N}$ be instantiated as an `add_monoid` using the operator `add := min`? Briefly explain your answer.

2e. Using the representation of p-adic numbers as left-infinite streams of digits, compute the following addition in base $p = 7$:

$$\begin{array}{r} \ldots 66666666 \\ +\qquad\qquad 1 \\ \hline \end{array}$$

Question 3. Even and odd (4+4+8 points)

Consider the following inductive definition of even numbers:

```
inductive even : ℕ → Prop
| zero             : even 0
| plus_two {n : ℕ} : even n → even (n + 2)
```

3a. Define a similar predicate for odd numbers, by completing the Lean definition below:

```
inductive odd : ℕ → Prop
```

The definition should distinguish two cases, like `even`, and should *not* rely on `even`.

3b. Give Lean proof terms for the propositions `odd 3` and `odd 5`, based on your answer to question 3a.

3c. Prove the following lemma by rule induction. Make sure to follow the guidelines given on page 1.

```
lemma even_odd {n : ℕ} (h : even n) :
  odd (n + 1)
```

Question 4. One-point rules (3+9 points)

One-point rules are lemmas that can be used to remove a quantifier from a proposition when the quantified variable can effectively take only one value. For example, $\forall$x, x = 6 $\to$ p x is equivalent to the much simpler p 6.

4a. Louis Reasoner proposes the following nonstandard one-point rule for $\exists$:

```
axiom exists.one_point_rule' {α : Type} {t : α} {p : α → Prop} :
  (∃x : α, x = t → p x) ↔ p t
```

What is wrong with this rule?

4b. Prove the following lemma:

```
lemma forall_exists.one_point_rule {α : Type} {t : α} {p : α → Prop} :
  (∀x : α, x = t → p x) ↔ (∃x : α, x = t ∧ p x)
```

In your proof, clearly identify how the quantifiers are instantiated.

Question 5. Arithmetic expressions (8+4 points)

Consider this simple type of arithmetic expressions:

```
inductive exp : Type
| var  : string → exp
| num  : ℤ → exp
| plus : exp → exp → exp

export exp (var num plus)
```

The following evaluation function computes the numeric value of an expression given an environment `env`:

```
def eval (env : string → ℤ) : exp → ℤ
| (var x)       := env x
| (num i)       := i
| (plus e₁ e₂) := eval e₁ + eval e₂
```

We want to rewrite arithmetic expressions such that `plus` is regrouped to the right. That is, every subexpression of the form `plus (plus e₁ e₂) e₃` should become `plus e₁ (plus e₂ e₃)`. The function `regroup`, which relies on the auxiliary function `shuffle`, performs this regrouping:

```
def shuffle : exp → exp → exp
| (var x)       a := plus (var x) a
| (num i)       a := plus (num i) a
| (plus e₁ e₂) a := shuffle e₁ (shuffle e₂ a)

def regroup : exp → exp
| e := shuffle e (num 0)
```

Note that `regroup` does not just regroup `plus` but also adds `0` at the end. Thankfully, this has no impact on the numeric value of the expression.

5a. Prove the following lemma about `shuffle` by structural induction. Make sure to follow the guidelines given on page 1.

```
lemma eval_shuffle (env : string → ℤ) (e a : exp) :
  eval env (shuffle e a) = eval env e + eval env a
```

5b. Prove the following lemma via a step-by-step calculational proof, with at most one rewrite rule or definition expansion per step, so that we can clearly see what happens. Your proof may rely on the lemma `eval_shuffle` from question 5a.

```
lemma eval_regroup (env : string → ℤ) (e : exp) :
  eval env (regroup e) = eval env e
```

Question 6. Big-step semantics (6+6+3+3 points)

On the occasion of this repeat exam, we introduce REPEAT, a brand-new programming language that resembles the WHILE language but whose defining feature is a `repeat` loop.

The Lean definition of its abstract syntax tree follows:

```
inductive program : Type
| skip {} : program
| assign  : string → (state → ℕ) → program
| seq     : program → program → program
| unless  : (state → Prop) → program → program
| repeat  : ℕ → program → program

export program (skip assign seq unless repeat)
```

The `skip`, `assign`, and `seq S T` statements have the same syntax and semantics as in the WHILE language. We also write `S ;; T` for `seq S T`.

The `unless b S` statement executes `S` unless `b` is true—i.e., it executes `S` if `b` is false. Otherwise, `unless b S` does nothing. In particular, `unless (λ_, true) S` is equivalent to `skip` (according to a big-step or a denotational semantics), and `unless (λ_, false) S` is equivalent to `S`. This construct is inspired by the Perl language.

The `repeat n S` statement executes `S` exactly `n` times. Thus, `repeat 10 S` is equivalent to `S ;; S ;; S ;; S ;; S ;; S ;; S ;; S ;; S ;; S`, and `repeat 0 S` is equivalent to `skip`.

6a. Complete the following specification of a big-step semantics as derivation rules.

$$\frac{}{(\mathtt{skip},\ \mathtt{s}) \Longrightarrow \mathtt{s}}\ \textsc{Skip} \qquad \frac{}{(\mathtt{assign\ x\ a},\ \mathtt{s}) \Longrightarrow \mathtt{s[x \mapsto s(a)]}}\ \textsc{Asn}$$

$$\frac{(\mathtt{S},\ \mathtt{s}) \Longrightarrow \mathtt{t} \quad (\mathtt{T},\ \mathtt{t}) \Longrightarrow \mathtt{u}}{(\mathtt{S\ ;;\ T},\ \mathtt{s}) \Longrightarrow \mathtt{u}}\ \textsc{Seq}$$

6b. Specify the same big-step semantics in Lean by completing the following definition:

```
inductive big_step : program × state → state → Prop
| skip {s} :
  big_step (skip, s) s
| assign {x a s} :
  big_step (assign x a, s) (s{x ↦ a s})
| seq {S T s t u} (h₁ : big_step (S, s) t) (h₂ : big_step (T, t) u) :
  big_step (S ;; T, s) u
```

6c. Is the REPEAT language deterministic? Briefly explain your answer.

6d. Are REPEAT programs guaranteed to terminate? Briefly explain your answer.

The grade for the exam is the total amount of points divided by 10, plus 1.